eval

wiki.tcl-lang.org/page/eval

eval , a built-in Tcl command, evaluates a script.

Synopsis

eval *arg* ?*arg ...*?

See Also

try
Can be used as byte-coded alternative to eval.

Documentation

official reference

Description

eval, short for uplevel 0, concatenates its arguments in the same fashion as concat, hands the result to the interpreter for evaluation as a script at the level of the caller of eval, and returns the result of the evaluation. This feature makes Tcl homoiconic. I.e. it is a **programmable programming language** which can be used to create **self-modifying programs**. Languages such as Lisp and Forth also have this feature.

if 1 ..., is an efficient equivalent to eval because it may be byte-compiled. eval itself may also be byte-compiled if it is called with only one argument.

Because the script is evaluated at the level of the caller of eval, a return causes the caller of eval to return.

eval is used by all the routines that receive scripts as arguments, including bind, everything with -command, etc.

When using eval, it is all too easy to leave holes which can be exploited to inject malicious code, so it is best to exercise extreme care with it.

There is a particular fact about a list that is relevant to eval: If a list is canonical, then it is also a script containing a single command, and no word in the command needs any substitution or escape processing. eval can only be sure that a list is canonical if it is a pure list. In the case of a pure list, eval can forego all parsing and substitution, and use the words in the list directly as the values of the command. Such a list can be built using the list commands.

Therefore, in order to avoid any double substitution or unintended interpretation of newline characters that might happen if the script wasn't a pure list, ensure that all arguments to eval are pure lists. In practice this usually means passing only one argument to eval. An additional benefit of calling eval with only pure lists is that eval does not generate the string representation of any argument. See (tcl objc type: args uses only string of own type , comp.lang.tcl, 2002-06-19). However, if even one argument is not a pure list, eval generates the string representation of every argument. This is often one of the big performance hits for those unaware of this consideration.

eval can often be avoided, particularly by using th {*} operator available in Tcl 8.5 or later.

Eval and double substitution

eval interprets the script it is given according to the rules of Tcl. This is a new round of interpretation, separate from the interpretation of the script that eval itself was part of. One must be prepared for this second round of interpretation.

Given the following assignment,

```
set b {the total is $20}
```

each of the following commands returns a different result:

```
set a $b
eval [list set a $b]
eval {set a $b}
eval "set a $b"
```

The words of the first command,

```
set a $b
```

are substituted before set is called. This substitution transforms the command into

```
set a {the total is $20}
```

At this point set is called with the arguments a, and the total is $20, and it sets the value of $a to the total is $20.

Likewise, the words of the second the command,

```
eval [list set a $b]
```

are substituted before eval is called. This substitution transforms the command into

```
eval {set a {the total is $20}}`
```

eval is then called with the script:

```
set a {the total is $20}
```

eval calls set with the arguments a {the total is $20}, and it sets the value of $a to the total is $20, as before.

In the third command,

```
eval {set a $b}
```

there are no substitutions to perform, so the command remains unchanged. eval then reads this script and substituts $b:

```
set a {the total is $20}
```

Finally, it calls set, which sets a to the total is $20.

In the final command,

```
eval "set a $b"
```

Tcl performs the substitutions within the double quotes, transforming the command to:

```
eval {set a the total is $20}
```

eval then interprets the script,

```
set a the total is $20
```

fails to find the variable $20 and raises an error. Even if there had ben no variable error, set would have been called with too many arguments, resulting in another error.

See double substitution for the details of why it can be dangerous.

Is eval Evil?

No, eval is not evil. It's just a matter of understanding that eval interprets a new script, and knowing how to properly quote things to achieve or avoid the desired interpretation.

Historically, eval was often used with exec to unpack lists, allowing a single list to provide multiple arguments to a command:

```
eval exec grep foo $filelist
```

so that *grep* receives multiple filename arguments rather than a single argument it wouldn't understand. As long as $filelist is a canonical list, i.e. uses only the space character to delimit words, this works.

In modern Tcl, the following equivalent command is preferred:

```
exec grep foo {*}$filelist
```

To correctly concatentate two lists:

```
set thislist [concat $thislist $thatlist] ;# Tcl 8.4
lappend thislist {*}$thatlist             ;# Tcl 8.5+
```

The following attempt does more than concatenate the lists: substitutes the value of the list into a script and then evaluates that script. I.e., any Tcl characters syntax such as $, [or \ are now interpreted according to their Tcl meanings.

```
# warning: this is not just simple list concatenation
eval lappend thislist $thatlist
```

The previous example is also an exammple of building up a script in pieces (by appending to a string) and finally calling

```
eval $script
```

Sometimes, such evaluation is desired. If it is not, use {*} to call the command without performing any further substitutions:

```
{*}$cmd
```

Unpacking Lists with eval

Prior to Tcl version 8.5, one use case for eval was to expand lists. There is no longer any reason to do this. Use {*} instead.

See Argument expansion.

Instead of:

```
eval pack [winfo children .]
```

use the equivalent command:

```
pack {*}[winfo children .]
```

And instead of:

```
set filepath [list path to the file]
eval file join $filepath
```

use:

```
set filepath [list path to the file]
file join {*}$filepath
```

Another similar example

```
#the old (bad) way
set f [glob *.tcl]
eval [linsert $f 0 exec lp]
```

```
#the shiny new way
exec lp {*}$f
```

Historical

The following describes the situation before the advent of {*}:

Naive code might look like this (an actual example from BWidget's entry.tcl:97):

```
#warning: bad code ahead!
eval entry $path $maps(:cmd)
```

An unsuccessful attempt to fix the problem:

```
#warning: bad code ahead!
eval entry [list $path] $maps(:cmd)
```

But $maps(:cmd) might be a string with newline characters:

```
set maps(:cmd) {
    -opt1 val1
    -opt2 val2
}
```

It's necessary to first convert $maps(:cmd) to a proper list using one of the list commands:

```
eval [linsert $maps(:cmd) 0 entry $path]
```

See Also

RE: TCLCORE Re: CFV: TIPs #156 and #157 , Jeff Hobbs, tcl-core mailing list, 2003-10-12
AMG: Text repeated below, please read it until you understand what trouble [eval] buys you and how helpful {*} is.

TCLCORE TIP #144: Argument Expansion Syntax , Jeff Hobbs, tcl-core mailing list, 2003-07-26

eval is evil (for spreading list-arguments) , comp.lang.tcl, 2003-03-11

Re: CFV: TIPs #156 and #157

Jeff Hobbs:

```
> miguel sofer <mig@ut...> writes:
> > In tcllib, every effort is done to provide code that runs
> > conditionally on the tcl version and provides the new functionality
> > toold interpreters.
>
> But the TIP doesn't specify any new functionality.  It only
> specifies a new syntax for functionality that we already
> have.  I don't see a need to complicate code by providing two
> implementations, when one of the implementations (the old
> one) works on all versions of Tcl and has no functional drawbacks.

Alright, this is where I step in to note how important this change is,
and to correct everyone's completely false assumption that the existing
eval hell has no functional drawbacks.  dgp asked me to post these
points that I made at Tcl2003 earlier, but I didn't think it necessary.
It obviously is.  It goes a little something like this:

Raise your hand if you think this is correct:

        eval entry $path $args

Everyone raising their hand please sit down.  You are wrong.  The $path
arg will be split apart as well, which is bad when using this in low
level code, like megawidgets or the like, where it must be handled
correctly.  After all, widgets with spaces in the names is 100% valid.

OK, so we know the fix, right?

        eval entry [list $path] $args

Ah, that's better ... but something is not right.  Hmmm ... oh, it is
inefficient!  The mix of list and string args will walk the wrong path
for optimization (this isn't important to everybody, but good low level
code writers should be sensitive to this).  OK, so that means this is
the best, right?

        eval [list entry $path] $args

Now I feel better.  What, that's not right?  If string args is actually
a multiline string, it won't work as expected.  Try it with:

        set args {
              -opt1 val1
              -opt2 val2
        }

and unfortunately that isn't theoretical.  I've seen code that uses a
$defaultArgs set up like that before regular args to handle defaults.

Ugh ... what are we left with?  This:

        eval [linsert $args 0 entry $path]

Only the final version is 100% correct, guaranteed not to blow when you
least want it to.
```

```
So we get back to the original point ... eval itself may not be
functionally flawed, but 99% of eval uses are.  In fact I don't always
use the final solution in code because it can get so unwieldly, but now
let me focus on tcllib, which was mentioned.  First let me say that my
favored solution is so much easier to use, has a minimal ugly factor,
and doesn't have any of the flaws above:

        entry $path {*}$args

So on to tcllib.  I just grep the .tcl files for "eval" and let me
pick a few:

# I sure hope critcl isn't in a dir with a space
./sak.tcl:        eval exec $critcl -force \
        -libdir [list $target] -pkg [list $pkg] $files

# Isn't this beautifully easy to understand?
./modules/ftp/ftp.tcl:        eval [concat $ftp(Output) {$s $msg $state}]

# I sure hope they don't use namespaces with spaces, or cmds ...
./modules/irc/irc.tcl:      eval [namespace current]::cmd-$cmd $args

# hmmm, just looks dangerous ...
./modules/struct/record.tcl:    eval Create $def ${inst_}.${inst} \
                                                [lindex $args $cnt_plus]

# I'm not sure why eval was used here.
# I think because version can be empty?
./modules/stooop/mkpkgidx.tcl:  eval package require $name $version

# I think someone needs to look up "concat"
./modules/textutil/adjust.tcl:  lappend list [ eval list $i $words($i) 1 ]

OK ... so I'm tired of looking now.

  Jeff Hobbs                     The Tcl Guy
  Senior Developer               http://www.ActiveState.com/
        Tcl Support and Productivity Solutions
```

Caveat: List-Like Strings

The arguments to eval are concatenated into a string to be interpreted, but this operation does not necessarily produce a syntactically-correct script. The following script breaks because the concatenation keeps the newline characters from the list's string representation, making eval interpret the second element as a new command:

```
% set command {a
b
c
}
a
b
c
% eval list $command
ambiguous command name "b": bgerror binary break
```

To solve this, construct the argument using list primitives like lappend, list, etc. DKF says: "list and eval are truly made for each other."

Another solution is to use the following idiom:

```
% eval [list {*}$command]
```

or equivalently,

```
% eval [linsert $command 0 list]
a b c
```

list produces a canonical list from its arguments, and linsert converts its first argument to a canoncial list, eliminating the newline characters in the process, and then inserts further elements into it (if any of these elements contain newlines characters, they remain in the resulting string).

It's important to remember that eval works on **strings**, not lists, and the rules for interpreting a string as a list are different than the rules for interpreting a string as a script.

Verbose Evaluation

LV: I just had to copy this over to the wiki - it is so neat!

From comp.lang.tcl, Bob Techentin writes in response to a poster:

It sounds like you're looking for something similar to /bin/sh "set -v", which prints shell input lines as they are read, and "set -x" which prints expanded commands. Kind of like a verbose mode.

Nope. Nothing like that. But you wouldn't have to write your own shell. You could walk through a script (a list of lines), and use info complete to decide how many lines are required to complete the command, then print both the command and the command's results. This seems to work.

```
proc verbose_eval script {
    set cmd {}
    foreach line [split $script \n] {
    if {$line eq {}} {continue}
        append cmd $line\n
        if {[info complete $cmd]} {
            puts -nonewline $cmd
            puts -nonewline [uplevel 1 $cmd]
            set cmd {}
        }
    }
}

set script {
    puts hello
    expr {2.2 * 3}
}

verbose_eval $script
```

LV: This proc is so slick! I love it!

Lars H: Another approach to this (which also can see individual commands in if branches, loop bodies, etc.) is to use traces. Do we have that written down anywhere?

AR: Here is my proposal with traces. Rem: noop is some kind of magic cookie.

```
proc eval_verbose Script {
    uplevel 1 "\# noop\n$Script"
}

proc eval_verbose_trace {command-string op} {
    if {[string match {uplevel 1 \{\# noop*} ${command-string}]} {
            puts "# ${command-string}"
    }
}

trace add execution eval_verbose enterstep eval_verbose_trace

eval_verbose {
    pwd
    puts A
    expr 12*3
}
```

PYK 2014-05-10: See also Scripted List, which does something similar.

Evaluating Code in a Scoped Manner

wtracy 2008-06-23: I want the commands I run in eval to have their own set of variables independently of the calling script. Is there some way I can hand eval a context (maybe as an associative array?).

Lars H: Sounds like you want a lambda (or possibly a closure, which is more difficult). If you're using Tcl 8.5, then have a look at apply.

RS: Long before 8.5, you could always write a little procedure to hide its local variables (x in this example):

```
% eval {proc {} x {expr {$x*$x}}; {} 5}
25
```

wtracy: Thanks. It looks like apply is the closest Tcl feature to what I want.

wtracy: Actually, it looks like what I *really* want is to nest eval inside of a namespace eval block.

wtracy: Okay, for the sake of any lost soul that comes along after me, this is what I really wanted to do all along:

```
$ set cmd {puts FOO}
puts FOO
$ namespace eval MyNameSpace $cmd
FOO
```

NEM: As mentioned, apply is probably a better choice in this case as it is less surprising wrt. variables. See dangers of creative writing for some related discussion. The apply version is:

```
apply {{} {puts FOO} ::MyNameSpace}
```

This has the benefit of evaluating the code within a fresh procedure context, meaning all variables are local

to that context by default. See also namespace inscope/namespace code and you may also like dict with.

Proposal: Modify eval to Accept a List of Lists

In some cases eval does work on lists - and its special. In particular, if eval is passed a pure list then it gets evaluated directly, without another round of substitution. However, I think the utility is just short of what it *could* be.

When you pass eval a pure list, you can only execute a single command. What if we were to pass eval a list of pure lists - it should directly evaluate each of them and return the value of the the last one evaluated. This sounds a lot like progn in lisp, and it seems like it would allow for some nifty bits of introspection - for example, if info body returned a pure list of lists rather than a string, that could be evaluated directly. Or modified. The whole program becomes a list of lists.

The problem is how to signal to eval (or uplevel, namespace, bind, ...) that it is a list of lists rather than just a list. Could eval determine if its input is a pure list and the first argument of that is a pure list, then evaluate as progn?

Another place this seems like it could be useful: I have a pkgIndex.tcl file with the line

```
package ifneeded tls 1.4 [
    list load [file join $dir libtls1.4.so]];[
    list source [file join $dir tls.tcl]]
```

where most of the lines are like

```
package ifneeded tclperl 2.3 [list load [file join $dir tclperl.so]]
```

since they only need one command; but the tls line needs two commands. So why can't it be

```
package ifneeded tls 1.4 [list [
    list load [file join $dir libtls1.4.so]] [
    list source [file join $dir tls.tcl]]]
```

Progn

RS 2004-02-06: As usual in Tcl, functionality you miss you can easy roll yourself. I needed a progn-like list eval in RPN again, and did it similar to this:

```
proc leval args {
    foreach arg $args {
        set res [uplevel 1 $arg]
    }
    set res ;# return last ("n-th") result, as Tcl evaluation does
}
```

eval versus bytecode

AMG: It appears eval'ed code does not get bytecode-compiled, even when eval is passed a single brace-quoted argument. The same is true for uplevel 0 and time. catch and if 1 seem to be bytecoded. {*} also gets bytecoded, but it doesn't accept a script "argument", rather a single command line pre-chewed into an objv list. Bytecoding cannot take place when the argument is the product of list, since the script isn't in a permanent Tcl_Obj.

```
proc a {} {eval           {for {set i 0} {$i < 100000} {incr i} {}}}
proc b {} {uplevel 0      {for {set i 0} {$i < 100000} {incr i} {}}}
proc c {} {time           {for {set i 0} {$i < 100000} {incr i} {}}}
proc d {} {catch          {for {set i 0} {$i < 100000} {incr i} {}}}
proc e {} {if {1}         {for {set i 0} {$i < 100000} {incr i} {}}}
proc f {} {            {*}{for {set i 0} {$i < 100000} {incr i} {}}}
proc g {} {eval      [list for {set i 0} {$i < 100000} {incr i} {}]}
proc h {} {uplevel 0 [list for {set i 0} {$i < 100000} {incr i} {}]}
proc i {} {time      [list for {set i 0} {$i < 100000} {incr i} {}]}
proc j {} {catch     [list for {set i 0} {$i < 100000} {incr i} {}]}
proc k {} {if {1}    [list for {set i 0} {$i < 100000} {incr i} {}]}
proc l {} {       {*}[list for {set i 0} {$i < 100000} {incr i} {}]}
proc m {} {try            {for {set i 0} {$i < 100000} {incr i} {}}}
a;b;c;d;e;f;g;h;i;j;k;l;m
time a 10    ;#  80723.8 microseconds per iteration - slow
time b 10    ;#  65380.2 microseconds per iteration - slow
time c 10    ;#  66024.8 microseconds per iteration - slow
time d 100   ;#  18888.3 microseconds per iteration - fast
time e 100   ;#  18779.3 microseconds per iteration - fast
time f 100   ;#  19375.2 microseconds per iteration - fast
time g 10    ;# 319111.5 microseconds per iteration - very slow
time h 10    ;# 342878.4 microseconds per iteration - very slow
time i 10    ;# 322279.2 microseconds per iteration - very slow
time j 10    ;# 316939.0 microseconds per iteration - very slow
time k 10    ;# 321865.5 microseconds per iteration - very slow
time l 10    ;# 344009.5 microseconds per iteration - very slow
time m 100   ;#  19503.0 microseconds per iteration - fast
```

I want single-argument eval functionality in my code, but I also want bytecoding. catch has the undesirable side effect of hiding errors, so I guess I have to use if 1 ... which is a really weird idiom. Does anyone have any better ideas?

AMG: I reran the tests and got better numbers. The current version of Tcl must be faster than whatever I used when I first did this benchmark (I'm using the same computer). Look in the page history to see the comparison. More importantly, I think I found a bytecoded eval: single-argument try. I added a test for it, and it turns out to be just as fast as the other "fast" methods. It's considerably less weird than if 1, and it doesn't hide errors.

DKF: We've been focusing a bit more on improving the most cripplingly-slow cases, but three execution modes still exist that have fundamentally different speeds. There's compilation to bytecode-with-local-var-table (which is the fastest; the speed comes from being able to compile in indexes into the LVT into the generated bytecode, which this test is particularly sensitive to), there's compilation to bytecode-without-LVT (slower; variables have to be looked up each time they're accessed), and there's interpreting (slowest by far). There's not much point in doing a lot of comparison between the three; they all exist for a reason. (We could make straight eval/uplevel 0 of constant arguments work at full speed, but we see no reason to bother given how rare they are in real code, and it's better that time is kept simple anyway.) You can usually tell what sort of class of speed you're going to get by using ::tcl::unsupported::disassemble to look at the bytecode to see whether variables are accessed by index or by name, or whether everything's done by just building arguments and invoking “cnormal” commands.

Twylite 2012-08-24: I've been optimising some control constructs and was trying to understand the performance of various combinations of uplevel, tailcall, catch and try. Along the way I found AMG's performance figures, and have updated them with some new ones:

```
# Fast (prefix f)
proc f_baseline       {} {                 for {set i 0} {$i < 100000} {incr i} {}
}
proc f_catch          {} {catch            {for {set i 0} {$i < 100000} {incr i}
{}}}
proc f_if_1           {} {if 1             {for {set i 0} {$i < 100000} {incr i}
{}}}
proc f_expand         {} {              {*}{for {set i 0} {$i < 100000} {incr i}
{}}}
proc f_try            {} {try              {for {set i 0} {$i < 100000} {incr i}
{}}}
proc f_time_apply     {} {time {apply {{}  {for {set i 0} {$i < 100000} {incr i}
{}}}}}

# Medium (prefix m)
proc m_eval           {} {eval             {for {set i 0} {$i < 100000} {incr i}
{}}}
proc m_uplevel_0      {} {uplevel 0        {for {set i 0} {$i < 100000} {incr i}
{}}}
proc m_time           {} {time             {for {set i 0} {$i < 100000} {incr i}
{}}}
proc m_tailcall_try   {} {tailcall    try  {for {set i 0} {$i < 100000} {incr i}
{}}}
proc m_catch_uplvl_1  {} {       set body  {for {set i 0} {$i < 100000} {incr i}
{}} ; catch { uplevel 1 $body } }
proc m_uplvl_1_catch  {} {       set body  {for {set i 0} {$i < 100000} {incr i}
{}} ; uplevel 1 [list catch $body] }

# Slow (prefix s)
proc s_eval_list      {} {eval       [list for {set i 0} {$i < 100000} {incr i}
{}]}
proc s_uplevel_0_list {} {uplevel 0  [list for {set i 0} {$i < 100000} {incr i}
{}]}
proc s_time_list      {} {time       [list for {set i 0} {$i < 100000} {incr i}
{}]}
proc s_catch_list     {} {catch      [list for {set i 0} {$i < 100000} {incr i}
{}]}
proc s_if_1_list      {} {if 1       [list for {set i 0} {$i < 100000} {incr i}
{}]}
proc s_expand_list    {} {        {*}[list for {set i 0} {$i < 100000} {incr i}
{}]}
proc s_tailcall       {} {tailcall         for {set i 0} {$i < 100000} {incr i} {}
}

if 0 {
    set REPS {f 10 m 40 s 100} ;# reps by prefix
    set cmds {f_baseline f_catch f_if_1 f_expand f_try f_time_apply
        m_eval m_uplevel_0 m_time m_tailcall_try m_catch_uplvl_1 m_uplvl_1_catch
        s_eval_list s_uplevel_0_list s_time_list s_catch_list s_if_1_list
        s_expand_list s_tailcall}

    foreach cmd $cmds {$cmd} ;# compile
    set cmdrepstimes {} ; foreach cmd $cmds { ;# time
        set reps [dict get $::REPS [string index $cmd 0]]
        lappend cmdrepstimes [lindex [time $cmd $reps] 0] $cmd $reps
    }
    set mintime [::tcl::mathfunc::min {*}$times]
    foreach {t cmd reps} [lsort -real -stride 3 $cmdrepstimes] {
        puts [format "time %-16s $reps\t;# %9.1f microseconds per iteration,
factor %5.2f" \
        $cmd $t [expr { $t / $mintime }] ]
    }
}
```

```
time f_time_apply     10        ;#    4625.8 microseconds per iteration, factor
1.00
time f_if_1           10        ;#    4641.8 microseconds per iteration, factor
1.00
time f_expand         10        ;#    4645.0 microseconds per iteration, factor
1.00
time f_catch          10        ;#    4651.3 microseconds per iteration, factor
1.00
time f_baseline       10        ;#    4658.5 microseconds per iteration, factor
1.01
time f_try            10        ;#    4735.9 microseconds per iteration, factor
1.02
time m_eval           40        ;#   18454.3 microseconds per iteration, factor
3.98
time m_uplevel_0      40        ;#   18738.3 microseconds per iteration, factor
4.04
time m_time           40        ;#   19176.8 microseconds per iteration, factor
4.14
time m_uplvl_1_catch  40        ;#   21501.1 microseconds per iteration, factor
4.64
time m_catch_uplvl_1  40        ;#   21603.1 microseconds per iteration, factor
4.66
time m_tailcall_try   40        ;#   21698.4 microseconds per iteration, factor
4.68
time s_uplevel_0_list 100       ;#   84963.3 microseconds per iteration, factor
18.34
time s_eval_list      100       ;#   85519.3 microseconds per iteration, factor
18.46
time s_catch_list     100       ;#   85919.9 microseconds per iteration, factor
18.54
time s_time_list      100       ;#   85944.2 microseconds per iteration, factor
18.55
time s_if_1_list      100       ;#   87269.0 microseconds per iteration, factor
18.84
time s_expand_list    100       ;#   93096.9 microseconds per iteration, factor
20.09
time s_tailcall       100       ;#   94473.5 microseconds per iteration, factor
20.39
```

History

eval is an old command that has been in Tcl from the beginning.